Java并发编程基础

2021年12月22日 16:49

线程简介

① 好处：多核CPU.
更快响应时间
更好编程模型

② 优先级. 1~10. 有些OS会忽略.
默认5.
多IO线程 优先级↑.
多CPU线程 优先级↓ 避免独占CPU.

③ 线程状态.

状态名称	说　明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕

状态变迁

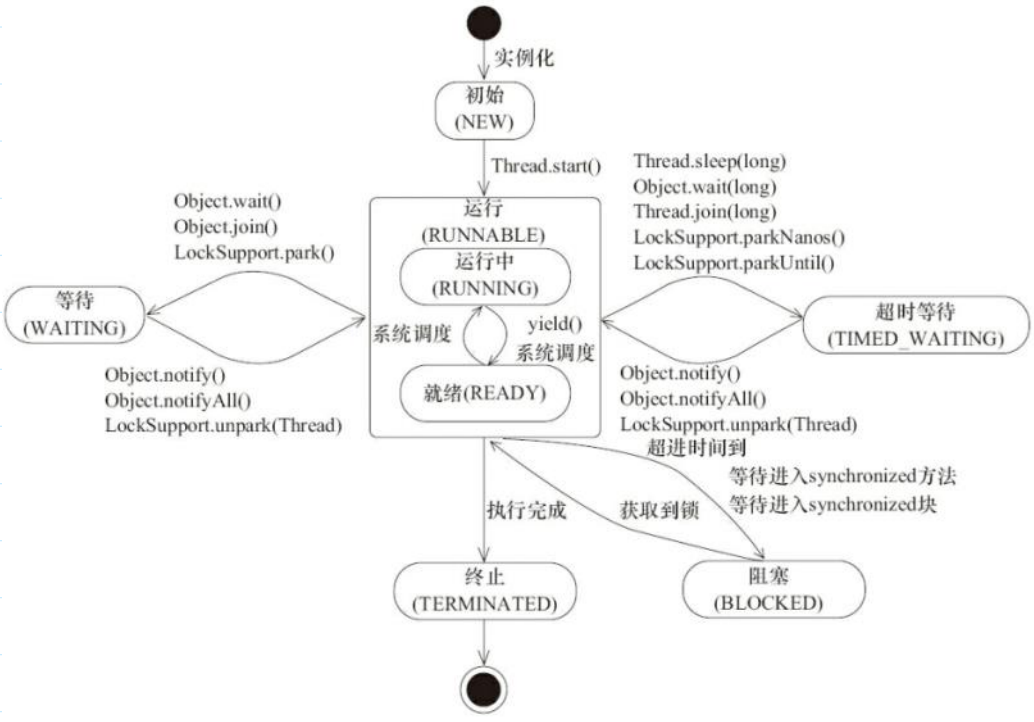

图4-1 Java线程状态变迁

④ Daemon线程.

启动和终止线程.

① 理解中断.

中断表示一个运行中的线程是否被其他线程调用了interrupt()方法.

Java API中抛出InterruptedException的方法:

在抛异常前,JVM会将线程的中断标识位清除.

② 过期的 suspend()、resume()、stop().

被禁用.
- suspend() 不会释放已占有资源.
- resume()
- stop() 不能保证资源正常释放.

采用等待/通知机制代替.

③ 安全地终止线程. 采用boolean值控制终止线程.

代码清单4-9 Shutdown.java

```
public class Shutdown {
    public static void main(String[] args) throws Exception {
        Runner one = new Runner();
        Thread countThread = new Thread(one, "CountThread");
        countThread.start();
        // 睡眠1秒，main线程对CountThread进行中断，使CountThread能够感知中断而结束
        TimeUnit.SECONDS.sleep(1);
        countThread.interrupt();
        Runner two = new Runner();
        countThread = new Thread(two, "CountThread");
        countThread.start();
        // 睡眠1秒，main线程对Runner two进行取消，使CountThread能够感知on为false而结束
        TimeUnit.SECONDS.sleep(1);
        two.cancel();
    }
    private static class Runner implements Runnable {
        private long i;
            private volatile boolean on = true;
            @Override
            public void run() {
            while (on && !Thread.currentThread().isInterrupted()){
                i++;
            }
            System.out.println("Count i = " + i);
        }
        public void cancel() {
            on = false;
        }
    }
}
```

输出结果如下所示(输出内容可能不同)。

```
Count i = 543487324
Count i = 540898082
```

示例在执行过程中，main线程通过中断操作和cancel()方法均可使CountThread得以终止。这种通过标识位或者中断操作的方式能够使线程在终止时有机会去清理资源，而不是武断地将线程停止，因此这种终止线程的做法显得更加安全和优雅。

线程间通信.

① synchronized原理.

Monitor.Enter 监视器 Monitor.Enter成功 对象 Monitor.Exit

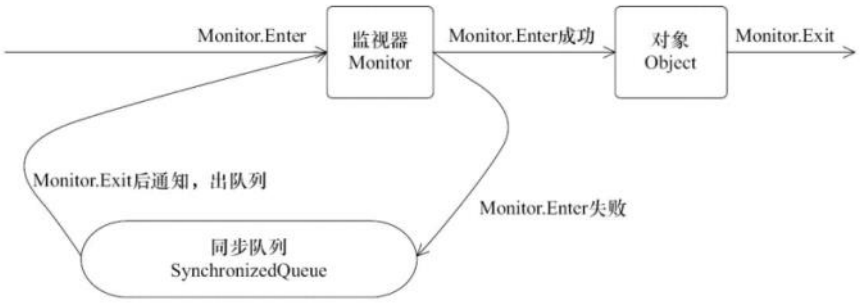

图4-2 对象、监视器、同步队列和执行线程之间的关系

从图4-2中可以看到，任意线程对Object（Object由synchronized保护）的访问，首先要获得Object的监视器。如果获取失败，线程进入同步队列，线程状态变为BLOCKED。当访问Object的前驱（获得了锁的线程）释放了锁，则该释放操作唤醒阻塞在同步队列中的线程，使其重新尝试对监视器的获取。

② 等待通知机制. notify()本质上是将线程从等待队列移动到同步队列.

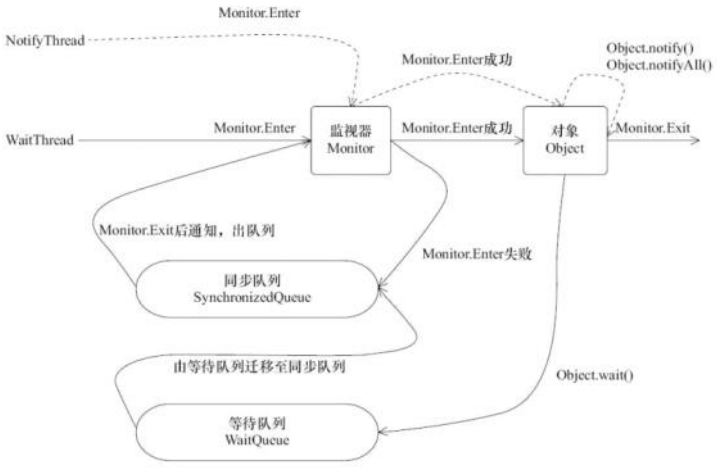

图4-3 WaitNotify.java运行过程

在图4-3中，WaitThread首先获取了对象的锁，然后调用对象的wait()方法，从而放弃了锁并进入了对象的等待队列WaitQueue中，进入等待状态。由于WaitThread释放了对象的锁，

NotifyThread随后获取了对象的锁，并调用对象的notify()方法，将WaitThread从WaitQueue移到SynchronizedQueue中，此时WaitThread的状态变为阻塞状态。NotifyThread释放了锁之后，WaitThread再次获取到锁并从wait()方法返回继续执行。

③ 等待/通知的经典范式.

等待方遵循如下原则。

1）获取对象的锁。

2）如果条件不满足，那么调用对象的wait()方法，被通知后仍要检查条件。

3）条件满足则执行对应的逻辑。

对应的伪代码如下。

```
synchronized(对象) {
        while(条件不满足) {
                对象.wait();
        }
        对应的处理逻辑
}
```

通知方遵循如下原则。

1）获得对象的锁。

2）改变条件。

3）通知所有等待在对象上的线程。

对应的伪代码如下。

```
synchronized(对象) {
        改变条件
        对象.notifyAll();
}
```

④ Thread.join().

↘ 等待该线程返回后，join()方法才返回.

④ Thread.join().

↘ 等待该线程返回后，join()方法才返回.

⑤ ThreadLocal.